

Практическое задание 2.

Продвинутые методы безусловной оптимизации.

Начало выполнения задания: 6 октября 2016 г.

Срок сдачи: 19 сентября (среда), 23:59.

Среда для выполнения задания: Python 3.

1 Модель двухклассовой логистической регрессии

Рассматривается задача классификации на два класса. Имеется обучающая выборка $\mathcal{D} := \{(x_i, y_i)\}_{i=1}^n$, где $x_i \in \mathbb{R}^d$ — вектор признаков i -го объекта, а $y_i \in \{-1, +1\}$ — его метка класса. Задача заключается в предсказании метки класса y_{new} для нового объекта, представленного своим вектором признаков x_{new} .

В модели логистической регрессии предсказание метки класса выполняется по знаку линейной функции:

$$y(x) := \text{sign}(x^\top w),$$

где $w \in \mathbb{R}^d$ — параметры модели, настраиваемые в процессе обучения.

Обучение модели осуществляется с помощью минимизации следующей функции потерь:

$$f(w) := \frac{1}{n} \sum_{i=1}^n \ln(1 + \exp(-y_i x_i^\top w)) + \frac{\lambda}{2} \|w\|_2^2. \quad (1)$$

Здесь $\lambda > 0$ — задаваемый пользователем коэффициент регуляризации. Использование регуляризации позволяет снизить вероятность переобучения алгоритма.

2 Метод L-BFGS

Рассматривается задача безусловной оптимизации

$$\min_{x \in \mathbb{R}^n} f(x),$$

где функция f является всюду непрерывно дифференцируемой.

Каждая итерация метода L-BFGS заключается в построении направления спуска d_k и шага в этом направлении:

$$x_{k+1} = x_k + \alpha_k d_k,$$

где $\alpha_k > 0$ — длина шага. Основная работа на итерации L-BFGS затрачивается на построение направления d_k . Для этого в методе сохраняется история $\mathcal{H}_k := \{(s_{k-i}, y_{k-i})\}_{i=1}^m$ из последних m (типичное значение $m = 10$) векторов¹

$$s_k := x_{k+1} - x_k,$$

$$y_k := \nabla f(x_{k+1}) - \nabla f(x_k).$$

По имеющейся истории вычисление направления d_k осуществляется с помощью следующего алгоритма:

¹При $k < m$ история \mathcal{H}_k состоит из k векторов.

```

1  $d \leftarrow -\nabla f(x_k);$ 
2 for  $i = k-1, \dots, k-m$  do
3    $\alpha_i \leftarrow (s_i^\top d)/(s_i^\top y_i);$ 
4    $d \leftarrow d - \alpha_i y_i;$ 
5 end
6  $d \leftarrow ((s_{k-1}^\top y_{k-1})/(y_{k-1}^\top y_{k-1}))d;$ 
7 for  $i = k-m, \dots, k-1$  do
8    $\beta \leftarrow (y_i^\top d)/(s_i^\top y_i);$ 
9    $d \leftarrow d + (\alpha_i - \beta)s_i;$ 
10 end
11 return  $d$ 

```

3 Неточный метод Ньютона

Рассматривается задача безусловной оптимизации

$$\min_{x \in \mathbb{R}^n} f(x), \quad (2)$$

где функция f является всюду дважды непрерывно дифференцируемой.

Метод Ньютона строит последовательность точек x_k , сходящуюся к решению задачи (2). Каждая итерация метода Ньютона имеет вид

$$x_{k+1} = x_k + \alpha_k d_k.$$

Направление шага d_k находится из следующей системы линейных уравнений:

$$H_k d_k = -\nabla f(x_k), \quad (3)$$

где $H_k = H_k^\top \succ 0$ — гессиан функции f в точке x_k (или его положительно определенная аппроксимация). Длина шага α_k выбирается с помощью поиска по прямой.

В неточном методе Ньютона система (3) решается с помощью метода сопряженных градиентов. В этом случае сама матрица H_k методу не нужна, а нужна только процедура умножения матрицы H_k на произвольный вектор $v \in \mathbb{R}^n$. Кроме того, не имеет большого смысла решать систему (3) сильно точно, если текущая точка x_k находится далеко от оптимума. Поэтому обычно метод сопряженных градиентов останавливают, как только невязка $r_k(d) := H_k d + \nabla f(x_k)$ удовлетворяет условию

$$\|r_k(d)\| \leq \eta_k \|\nabla f(x_k)\|. \quad (4)$$

Последовательность $\{\eta_k\}$, $\eta_k \in (0, 1)$, называется *форсирующей последовательностью* и обычно выбирается следующим образом:

$$\eta_k := \min \left\{ 0.5, \sqrt{\|\nabla f(x_k)\|} \right\}.$$

Такой выбор η_k гарантирует суперлинейную скорость сходимости метода.

Следует отметить, что ранний выход из метода сопряженных градиентов по условию (4), вообще говоря, не гарантирует того, что найденное направление d_k будет направлением спуска функции f в точке x_k (в отличие от точного решения $-H_k^{-1}\nabla f(x_k)$ системы (3)). Поэтому после выхода из метода сопряженных градиентов нужно дополнительно проверять, удовлетворяет ли d_k условию $d_k^\top g_k > 0$, и если нет, то снова запустить метод сопряженных градиентов, но теперь уже из начальной точки d_k и с меньшим значением η_k (например, уменьшить η_k в 10 раз). Такую процедуру нужно повторять до тех пор, пока d_k не станет направлением спуска (что гарантированно рано или поздно произойдет).

4 Формулировка задания

1. Реализовать стандартный метод сопряженных градиентов для решения системы линейных уравнений $Ax = b$ для симметричной положительно определенной матрицы $A = A^\top \succ 0$.
2. Исследовать скорость сходимости реализованного алгоритма в зависимости от числа обусловленности $\kappa := \lambda_{\max}(A)/\lambda_{\min}(A)$ матрицы системы. Для этого выбрать несколько сильно отличающихся значений κ и для каждого из них:
 - (a) Сгенерировать несколько случайных матриц A с собственными значениями, равномерно распределенными на интервале $[1, \kappa]$, и случайных правых частей b .
 - (b) Построить графики сходимости метода сопряженных градиентов (ℓ_∞ -норма невязки против числа итераций) для каждой сгенерированной пары (A, b) .

Все кривые сходимости нужно рисовать на одном графике. Для каждого фиксированного κ все кривые должны иметь одинаковый цвет, для разных κ используются разные цвета. В итоге должен получиться **один** график с набором кривых для каждого κ .

Рекомендация: Случайную симметричную матрицу $A \in \mathbb{R}^{n \times n}$ с заданным спектром $s \in \mathbb{R}^n$ можно сгенерировать, используя спектральное разложение:

- (a) Сгенерировать случайную ортогональную матрицу Q . Это можно сделать с помощью команды `sp.linalg.orth(np.random.randn(n, n))`.
- (b) Положить $A := QSQ^\top$, где $S := \text{Diag}(s)$.

Рекомендация: Чтобы получить красивую картинку, лучше рисовать кривые одного цвета с прозрачностью. Для этого можно использовать параметр `alpha` в функциях типа `plt.plot` (например, `alpha=0.8`).

3. Реализовать следующие процедуры для логистической регрессии:
 - (a) Процедура вычисления значения функции потерь $f(w)$ и ее градиента $\nabla f(w)$.
 - (b) Процедура умножения гессиана $\nabla^2 f(w)$ на произвольный вектор v .

Реализация каждой процедуры должна быть векторизованной, т. е. код не должен содержать никаких циклов. Также запрещается формировать в памяти какие-либо дополнительные матрицы (например, диагональные). Написанный код должен корректно работать как с плотными данными, заданными в обычном формате `numpy`-массива, так и с разреженными данными, заданные в формате разреженной `scipy`-CSR-матрицы.

4. Проверить правильность реализации подсчета градиента и действия гессиана на вектор с помощью конечных разностей:

$$\begin{aligned} [\nabla f(x)]_i &\approx \frac{f(x + \epsilon_1 e_i) - f(x)}{\epsilon_1}, \\ [\nabla^2 f(x)v]_i &\approx \frac{f(x + \epsilon_2 v + \epsilon_2 e_i) - f(x + \epsilon_2 v) - f(x + \epsilon_2 e_i) + f(x)}{\epsilon_2^2} \end{aligned} \quad (5)$$

где $e_i := (0, \dots, 0, 1, 0, \dots, 0)$ — i -й базисный орт, а ϵ_1, ϵ_2 — достаточно маленькие положительные числа: $\epsilon_1 \sim \sqrt{\epsilon_{\text{mach}}}$, $\epsilon_2 \sim \sqrt[3]{\epsilon_{\text{mach}}}$, где ϵ_{mach} — машинная точность ($\approx 10^{-16}$ для типа `double`). Для этого нужно сгенерировать небольшую модельную выборку \mathcal{D} и проверить выполнение соотношений (5) для всех i в нескольких пробных точках x и v .

5. Реализовать следующие методы:

- (a) Нелинейный метод сопряженных градиентов (версия Дай-Юань);
- (b) Метод L-BFGS;
- (c) Неточный метод Ньютона.

Для выбора длины шага во всех методах использовать сильные условия Вульфа.

Рекомендация: Для поиска точки, удовлетворяющей сильным условиям Вульфа, можно использовать библиотечную функцию `line_search_wolfe2` из модуля `sp.optimize.linesearch`. Однако следует отметить, что в `scipy` используется немного другой интерфейс для работы с оптимизируемой функцией: предполагается, что подсчет значения функции и градиента реализован в разных процедурах, а не в одной (как в этом задании). Поэтому для запуска `line_search_wolfe2` внутри самостоятельно реализованного метода оптимизации, принимающего на вход `func` — процедуру, вычисляющую значение функции и ее градиент, — нужно использовать код следующего вида:

```
func_f = lambda x: func(x)[0]
func_g = lambda x: func(x)[1]
alpha = line_search_wolfe2(func_f, func_g, x, d, g, c1=c2, c2=c2)[0]
```

Однако, несмотря на свою простоту, у такого решения есть существенный недостаток: поскольку внутри `line_search_wolfe2` практически в каждой точке вызываются и `func_f`, и `func_g`, то в итоге процедура `func` будет вызвана в два раза больше раз, чем могла бы (будь в `scipy` другой интерфейс). Чтобы устранить этот недостаток, следует дополнительно реализовать специальный класс-обертку, который будет запоминать внутри себя последнюю точку `x_last`, в которой была вызвана процедура `func`, а также выход `out_last` этой процедуры; при каждом последующем вызове `func(x)` будет сначала проверяться, совпадают ли `x` и `x_last`, и если да, то вместо вызова `func(x)` будет возвращено `out_last`, а если нет, то будет выполнен вызов `func(x)` и перезаписаны значения `x_last` и `out_last`:

```
class FuncWrapper:
    def __init__(self, func):
        self.func = func
        self.x_last = None
        self.out_last = None
    def __call__(self, x):
        if self.x_last is None or not np.all(self.x_last == x):
            self.out_last = self.func(x)
            self.x_last = np.copy(x)
        return self.out_last
```

Имея в распоряжении класс `FuncWrapper`, можно написать следующий код для вызова процедуры `line_search_wolfe2`, который теперь будет выполнять лишь один вызов `func` в каждой точке:

```
func_wrapper = FuncWrapper(func)
func_f = lambda x: func_wrapper(x)[0]
func_g = lambda x: func_wrapper(x)[1]
alpha = line_search_wolfe2(func_f, func_g, x, d, g, c1=c2, c2=c2)[0]
```

6. Сравнить три реализованных метода на задаче обучения двухклассовой логистической регрессии на реальных данных. Рассмотреть при этом ситуации малого числа переменных ($d \lesssim 300$) и большого ($d \gtrsim 5000$)². Построить графики сходимости следующих двух видов: 1) невязка (по функции) против числа вызовов оракула; 2) невязка (по функции) против реального времени работы. Коэффициент регуляризации и начальную точку взять стандартным образом: $\lambda = 1/n$, $w_0 = 0$.

²Для каждой из этих ситуаций рассмотреть, в идеале, (как минимум) 2-3 разных набора данных.

Рекомендация: Реальные данные можно взять с сайта LIBSVM³. Любой набор данных с сайта LIBSVM представляет из себя текстовый файл в формате svmlight. Чтобы считать такой текстовый файл, можно использовать функцию `load_svmlight_file` из модуля `sklearn.datasets`.

7. Написать отчет в формате PDF с описанием всех проведенных исследований.

5 Оформление задания

Результатом выполнения задания являются 1) pdf-отчет о проведенных исследованиях, 2) zip-архив, содержащий все необходимые исходные коды: `optim.py`, `lossfuncs.py` и `special.py`. Выполненное задание следует отправить письмом по адресу `bayesml@gmail.com` с заголовком

«[BMK MOMO16] Задание 2, Фамилия Имя».

Убедительная просьба присылать выполненное задание только один раз с окончательным вариантом.

График сходимости алгоритма оптимизации должен показывать зависимость невязки от числа вызовов оракула (или реального времени работы). Во всех графиках должна использоваться логарифмическая шкала для невязки. При сравнении различных методов на одной и той же задаче кривые сходимости этих методов нужно рисовать на одном графике; таким образом, каждый график должен соответствовать отдельной задаче и содержать несколько кривых, соответствующих разным алгоритмам.

Поскольку проверка реализованных алгоритмов будет осуществляться в полуавтоматическом режиме, все реализованные функции должны строго соответствовать приведенным ниже прототипам и корректно запускаться в Python 3 (а не Python 2!). Проверить наличие всех необходимых функций, а также их соответствие требуемым прототипам можно с помощью специального скрипта `check_submission_ass2.py`, выдаваемого вместе с текстом задания.

6 Прототипы функций

1. Метод сопряженных градиентов для решения системы линейных уравнений $Ax = b$:

Модуль:	<code>optim</code>
Функция:	<code>cg(matvec, b, x0, tol=1e-4, max_iter=None, disp=False, trace=False)</code>
Параметры:	<code>matvec: callable matvec(x)</code> Функция умножения матрицы системы на произвольный вектор x . Принимает: <code>x: np.ndarray</code> Вектор размера n . Возвращает: <code>ax: np.ndarray</code> Произведение матрицы системы на вектор x , вектор размера n . <code>b: np.ndarray</code> Правая часть системы, вектор размера n . <code>x0: np.ndarray</code> Начальная точка, вектор размера n . <code>tol: float</code> , опционально Точность по ℓ_∞ -норме невязки: <code>norm(A x_k - b, inf) <= tol</code> .

³<http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>.

	<p>max_iter: <code>int</code> или <code>None</code>, опционально Максимальное число итераций метода. Если <code>None</code>, то положить равным n.</p> <p>disp: <code>bool</code>, опционально Отображать прогресс метода по итерациям (номер итерации, текущая норма невязки и пр.) или нет.</p> <p>trace: <code>bool</code>, опционально Сохранять траекторию метода для возврата истории или нет.</p>
Возврат:	<p>x_sol: <code>np.ndarray</code> Найденное решение системы, вектор размера n.</p> <p>status: <code>int</code> Статус выхода, число: 0: решение найдено с заданной точностью <code>tol</code>; 1: достигнуто максимальное число итераций.</p> <p>hist: <code>dict</code>, возвращается только если <code>trace=True</code> История процесса оптимизации по итерациям. Словарь со следующими полями:</p> <p>norm_r: <code>np.ndarray</code> Норма невязки $\ Ax_k - b\ _\infty$ по итерациям.</p>

2. Оракул функции потерь логистической регрессии (1):

Модуль:	<code>lossfuncs</code>
Функция:	<code>logistic(w, X, y, reg_coef)</code>
Параметры:	<p>w: <code>np.ndarray</code> Точка вычисления, вектор размера d.</p> <p>X: <code>np.ndarray</code> или <code>sp.sparse.csr_matrix</code> Матрица признаков размеров $n \times d$.</p> <p>y: <code>np.ndarray</code> Метки классов, вектор размера n, состоящий из элементов $\{-1, +1\}$.</p> <p>reg_coef: <code>float</code> Коэффициент регуляризации $\lambda > 0$.</p>
Возврат:	<p>f: <code>float</code> Значение функции в точке w.</p> <p>g: <code>np.ndarray</code> Градиент функции в точке w, вектор размера d.</p>

3. Умножение гессиана функции логистической регрессии (1) на произвольный вектор:

Модуль:	<code>lossfuncs</code>
Функция:	<code>logistic_hess_vec(w, v, X, y, reg_coef)</code>
Параметры:	<p>w: <code>np.ndarray</code> Точка вычисления, d-мерный вектор.</p> <p>v: <code>np.ndarray</code></p>

	<p>Вектор, на который умножается гессиан, d-мерный вектор.</p> <p>X: <code>np.ndarray</code> или <code>sp.sparse.csr_matrix</code></p> <p>Матрица признаков размеров $n \times d$.</p> <p>y: <code>np.ndarray</code></p> <p>Метки классов, вектор размера n, состоящий из элементов $\{-1, +1\}$.</p> <p>reg_coef: <code>float</code></p> <p>Коэффициент регуляризации $\lambda > 0$.</p>
Возврат:	<p>hv: <code>np.ndarray</code></p> <p>Вектор $\nabla^2 f(w)v$.</p>

4. Подсчет градиента и действия гессиана на вектор с помощью конечных разностей:

Модуль:	<code>special</code>
Функция:	<code>grad_finite_diff(func, x, eps=1e-8)</code>
Параметры:	<p>func: <code>callable func(x)</code></p> <p>Функция, градиент которой нужно вычислить.</p> <p>Принимает:</p> <p>x: <code>np.ndarray</code></p> <p>Аргумент функции, вектор размера n.</p> <p>Возвращает:</p> <p>f: <code>float</code></p> <p>Значение функции в точке x.</p> <p>x: <code>np.ndarray</code></p> <p>Точка, в которой нужно вычислить градиент, вектор размера n.</p> <p>eps: <code>float</code>, опционально</p> <p>Число ϵ_1 в формуле (5).</p>
Возврат:	<p>g: <code>np.ndarray</code></p> <p>Оценка градиента по формуле (5), n-мерный вектор.</p>

Модуль:	<code>special</code>
Функция:	<code>hess_vec_finite_diff(func, x, v, eps=1e-5)</code>
Параметры:	<p>func: <code>callable func(x)</code></p> <p>Функция, у которой нужно вычислить произведение гессиана на вектор.</p> <p>Принимает:</p> <p>x: <code>np.ndarray</code></p> <p>Аргумент функции, вектор размера n.</p> <p>Возвращает:</p> <p>f: <code>float</code></p> <p>Значение функции в точке x.</p> <p>x: <code>np.ndarray</code></p> <p>Точка, в которой нужно вычислить гессиан, n-мерный вектор.</p> <p>v: <code>np.ndarray</code></p>

	<p>Вектор, на который нужно умножить гессиан; размер вектора n.</p> <p><code>eps: float</code>, опционально</p> <p>Число ϵ_2 в формуле (5).</p>
Возврат:	<p><code>hv: np.ndarray</code></p> <p>Оценка $\nabla^2 f(x)v$ по формуле (5), вектор размера n.</p>

5. Нелинейный метод сопряженных градиентов:

Модуль:	<code>optim</code>
Функция:	<code>ncg(func, x0, tol=1e-4, max_iter=500, c1=1e-4, c2=0.1, disp=False, trace=False)</code>
Параметры:	<p><code>func: callable func(x)</code></p> <p>Оракул минимизируемой функции.</p> <p>Принимает:</p> <p><code>x: np.ndarray</code></p> <p>Точка вычисления, вектора размера n.</p> <p>Возвращает:</p> <p><code>f: float</code></p> <p>Значение функции в точке x.</p> <p><code>g: np.ndarray</code></p> <p>Градиент функции в точке x, вектора размера n.</p> <p><code>x0: np.ndarray</code></p> <p>Начальная точка, вектор размера n.</p> <p><code>tol: float</code>, опционально</p> <p>Точность по ℓ_∞-норме градиента: <code>norm(nabla f(x_k), infy) < tol</code>.</p> <p><code>max_iter: int</code>, опционально</p> <p>Максимальное число итераций метода.</p> <p><code>c1: float</code>, опционально</p> <p>Значение константы c_1 в условиях Вульфа.</p> <p><code>c2: float</code>, опционально</p> <p>Значение константы c_2 в условиях Вульфа.</p> <p><code>disp: bool</code>, опционально</p> <p>Отображать прогресс метода по итерациям (номер итерации, число вызовов оракула, значение функции, норма градиента и пр.) или нет.</p> <p><code>trace: bool</code>, опционально</p> <p>Сохранять траекторию метода для возврата истории или нет.</p>
Возврат:	<p><code>x_min: np.ndarray</code></p> <p>Найденная точка минимума, вектор размера n.</p> <p><code>f_min: float</code></p> <p>Значение функции в точке <code>x_min</code>.</p> <p><code>status: int</code></p> <p>Статус выхода, число:</p> <p>0: решение найдено с заданной точностью <code>tol</code>;</p>

	1: достигнуто максимальное число итераций или вызовов оракула. hist : dict, возвращается только если trace=True История процесса оптимизации по итерациям. Словарь со следующими полями: f : np.ndarray Значение функции. norm_g : np.ndarray ℓ_∞ -норма градиента. n_evals : np.ndarray Суммарное число вызовов оракула на текущий момент. elaps_t : np.ndarray Реальное время, пройденное с начала оптимизации.
--	---

6. (L-BFGS) Процедура нахождения направления d_k :

Модуль:	optim
Функция:	lbfgs_compute_dir (sy_hist, g)
Параметры:	sy_hist : collections.deque История \mathcal{H}_k , структура данных «дек». Каждый элемент sy_hist [-i] (т. е. i -й справа) — это пара (s_{k-i}, y_{k-i}) из двух элементов типа np.ndarray размера n . g : np.ndarray Градиент $\nabla f(x_k)$ в текущей точке x_k , n -мерный вектор.
Возврат:	d : np.ndarray Направление спуска метода L-BFGS, вектор размера n .

7. Прототипы функции **lbfgs** для метода LBFGS полностью повторяет прототип функции **nsg** за исключением следующих двух пунктов:

- (a) В **lbfgs** имеется дополнительный опциональный параметр **m**, указывающий размер истории, используемой методом (т.е. число хранимых пар (s_k, y_k)). По умолчанию **m=10**.
- (b) В **lbfgs** по умолчанию **c2=0.9**.

8. Неточный метод Ньютона:

Модуль:	optim
Функция:	hfn (func, x0, hess_vec, tol=1e-4, max_iter=500, c1=1e-4, c2=0.9, disp=False, trace=False)
Параметры:	func : callable func(x) Оракул минимизируемой функции. Принимает: x : np.ndarray Точка вычисления, вектор размера n . Возвращает: f : float Значение функции в точке x .

	<p><code>g: np.ndarray</code> Градиент функции в точке <code>x</code>, вектор размера <code>n</code>.</p> <p><code>x0: np.ndarray</code> Начальная точка, вектор размера <code>n</code>.</p> <p><code>hess_vec: callable hess_vec(x, v)</code> Функция умножения гессиана (или его аппроксимации) в точке <code>x</code> на произвольный вектор <code>v</code>. Принимает: <code>x: np.ndarray</code> Точка вычисления, вектор размера <code>n</code>. <code>v: np.ndarray</code> Вектор для умножения размера <code>n</code>. Возвращает: <code>hv: np.ndarray</code> Произведение гессиана на вектор <code>v</code>, вектор размера <code>n</code>.</p> <p><code>tol: float</code>, опционально Точность по ℓ_∞-норме градиента: <code>norm(nabla f(x_k), infy) < tol</code>.</p> <p><code>max_iter: int</code>, опционально Максимальное число итераций метода.</p> <p><code>c1: float</code>, опционально Значение константы c_1 в условиях Вульфа.</p> <p><code>c2: float</code>, опционально Значение константы c_2 в условиях Вульфа.</p> <p><code>disp: bool</code>, опционально Отображать прогресс метода по итерациям (номер итерации, число вызовов орakкула, значение функции, норма градиента и пр.) или нет.</p> <p><code>trace: bool</code>, опционально Сохранять траекторию метода для возврата истории или нет.</p>
Возврат:	<p><code>x_min: np.ndarray</code> Найденная точка минимума, вектор размера <code>n</code>.</p> <p><code>f_min: float</code> Значение функции в точке <code>x_min</code>.</p> <p><code>status: int</code> Статус выхода, число: 0: решение найдено с заданной точностью <code>tol</code>; 1: достигнуто максимальное число итераций.</p> <p><code>hist: dict</code>, возвращается только если <code>trace=True</code> История процесса оптимизации по итерациям. Словарь со следующими полями: <code>f: np.ndarray</code> Значение функции. <code>norm_g: np.ndarray</code> ℓ_∞-норма градиента.</p>

`n_evals: np.ndarray`

Суммарное число вызовов оракула (сумма `func` и `hess_vec`) на текущий момент.

`elaps_t: np.ndarray`

Реальное время, пройденное с начала оптимизации.